Formal Verification and Automated Testing for Diagnostic and Monitoring Systems

Bruno Dutertre*, John Rushby†, and Ashish Tiwari*
SRI International, Menlo Park CA USA
César Muñoz‡and Radu Siminiceanu§
National Institute of Aerospace, Hampton VA USA

Formal methods offer the unique benefit that it is possible to examine all possible circumstances within a given scope, rather than merely sample them as with testing and simulation. This is possible even when huge numbers of discrete possibilities must be considered, such as in fault scenarios, and in the presence of real-time and continuous behavior, and when both these sources of difficulty are combined. Formal methods achieve this by using symbolic methods of analysis, and it is the computational complexity of these symbolic methods that limits and complicates their application.

Recently, new methods and tools for symbolic analysis have been developed that are far more efficient and effective in practice than earlier methods. These include solvers for "satisfiability modulo theories" (SMT solvers) and methods for using these to analyze discrete and hybrid (i.e., mixed discrete and continuous) systems. In addition, new ways to apply these capabilities have been developed, such as their use for test generation, and controller synthesis, in addition to analysis.

In this paper, we outline these new methods, and describe a project to extend and apply them in combination to issues in verification and testing of diagnostic and monitoring systems.

Nomenclature

`vxa`	Velocity of aircraft a in x dimension
`vxb`	Velocity of aircraft b in x dimension
`vxb`	Velocity of aircraft b in y dimension
`vya`	Velocity of aircraft a in y dimension

I. Introduction

Formal methods are approaches to the design and analysis of computer-based systems and software that employ mathematical techniques. They directly correspond to the mathematical techniques used in other fields of engineering, such as finite-element analysis for bridges and other structures, or computational fluid dynamics for airplane wings and streamlined cars.

Each field of engineering develops ways to model its designs and the phenomena of concern in some branch of mathematics so that calculation can be used to predict and explore the properties and behaviors of designs prior to construction. Most traditional engineering fields deal with physical systems, so the appropriate branches of mathematics are typically based on differential equations. Computers, on the other hand, deal with symbols: their hardware and software set up patterns of 0s and 1s that represent some state

*Senior Computer Scientist, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025
†Program Director, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025, Member AIAA
‡Senior Staff Scientist, NIA, 100 Exploration Way, Hampton VA 23666, Member AIAA
§Staff Scientist, NIA, 100 Exploration Way, Hampton VA 23666

of affairs (e.g., the numbers 5 and 20), and perform operations on these symbols that yield a representation of some new state of affairs (e.g., the control signals to be sent to an actuator that will move the flaps on an aircraft wing from their current setting of 5° to a desired setting of 20°). The branch of mathematics that deals with symbols and their manipulation and their relation to the world is formal logic, and this is the origin of the "formal" in formal methods.

Automated calculation in formal logic—that is to say, automated deduction, or theorem proving—developed more slowly than calculation in traditional engineering mathematics. In part this was because many researchers in the field had different agendas than raw calculational efficiency—they were interested, for example, in using theorem proving in the teaching of logic, or in reproducing mathematical proofs, or in modeling cognition for Artificial Intelligence—and in part it was because calculation in formal logic is intrinsically harder than most numerical mathematics: almost all computational problems in formal logic are at least NP-hard, many are superexponential, and some are undecidable. An intuitive reason why many of these problems are so hard is that they deal with discrete choices—corresponding to the `if-then-else` and iterative control structures that give computers their power—and the total number of possible combinations that has to be considered is exponential in the number of choices. But this exponential is also the reason why software is so hard to get right, and why testing is not very effective: a feasible number of tests examines only a tiny fraction of the possible number of different behaviors and, because behavior involving discrete choices is discontinuous, there is no basis for extrapolating from examined to unexamined cases. The attraction of modern formal methods is that, despite the intrinsic computational complexity, they often can perform effective calculations and thereby examine *all possible* behaviors.

The remainder of this paper introduces some of the ways in which formal methods can be applied in software and systems engineering, outlines the technology that underlies these applications, reports on some industrial experience in aviation systems, and sketches some prospects for the future.

II. Formal Methods

Static analysis is probably the most straightforwardly effective application of automated formal methods. Programmers are familiar with the typechecking performed by a Java or Ada compiler, and the rather weaker checking performed in C; the Java and Ada typecheckers catch errors such as adding a number to a Boolean, and even C will protest if an array is added to a number. Typechecking by a compiler is actually a simple kind of formal analysis but, because it is expected to operate in linear time, and to deliver no false alarms, its power is quite limited. If we are prepared to use methods of formal calculation that are potentially more expensive, then we can extend the range of properties that can be checked. For example, a static analyzer can warn about programs that might generate runtime exceptions due to division by zero, array bounds violation, or dereferencing a null pointer. This is kind of analysis is called "static" because it does not depend on actually running the program; testing a program by running it is called "dynamic" analysis. The advantage of static over dynamic analysis is that the former examines all possible executions, whereas the latter examines only the cases actually run.

It is difficult to make static analysis both automatic and precise, so we generally tolerate some false alarms, and may also allow some errors to go undetected. The balance between these kinds of imprecisions can be adjusted according to the needs of the application: for debugging, we may allow some real errors to go undetected in return for few false alarms, while for certification we tolerate more false alarms in return for guaranteed detection of all real errors in the analyzed class.

False alarms and undetected errors can arise for several reasons. One is related to the power of the theorem proving involved. Suppose for example, that a static analyzer looking for possible divide by zero exceptions encounters the expression $q/(z+2)$ in a context where $z = 3 \times x + 6 \times y - 1$, and x and y are arbitrary integers. A theorem prover that is competent for integer linear arithmetic will easily establish that $z + 2$ is nonzero under these constraints, whereas one that lacks special knowledge about integers will be unable to do so and will generate a false alarm. A different source of difficulty lies in programming constructs such as heap structures and pointers; static analyzers may maintain only abstract properties of heap objects (e.g., their "shape," such as whether they are a list or a tree) and will raise a warning for any division operation whose denominator comes from the heap. Much of the research and recent progress in formal methods addresses these difficulties, but there are limits to what can be achieved without additional information from the programmer.

There are two kinds of information that a programmer can supply to a static analyzer: information that tells it what to check for, and information that helps it do its analysis. One fairly unobtrusive way to accomplish both of these is through extended type annotations: instead of declaring a variable to be simply an integer, we could specify that it is a `nonzero` integer, and we could further specify that an array contains `nonzero` integers, or that it is `sorted`. The analyzer must check that the extended type annotations are never violated, and it can then use them as assumptions in checking other properties. For example, if the type annotation for w declares it to be an `even` integer, then it becomes trivial to verify that $p/(w+1)$ will not raise a division by zero exception.

Properties specified by types are *invariants*: they are required to hold throughout execution; simple assertions can be used to specify properties that must hold whenever execution reaches a given location, and technically these are also invariants (of the form "control at given location implies property"). Properties that are not invariants can be specified by alternative techniques that include those based on state machines, regular expressions, typestate annotations (in which allowed types can depend on the state), and temporal logic. These permit, for example, specification of the requirement that the lock on a resource should be alternately `claim`ed and `released`: two `claim`s in succession is an error. Because it considers all possible executions, static analysis for invariants and these other kinds of properties often reveals bugs in otherwise well-tested programs.

Formal methods and testing are not antithetic, however. Formal methods may assume properties of the environment (e.g., that the compiler for the programming language is correct, just as structural mechanics assumes properties of steel beams) that might be violated in particular cases—and testing in the real environment may reveal those violations. The specifications and methods of formal calculation that underlie static analysis also can be used in support of testing, thereby extending the range of benefits of formal methods. For example, specifications can be compiled into runtime monitors that check for violations and invoke exception handling. The benefits over conventional testing practice are that the monitored properties are synthesized from specifications rather than programmed by hand (e.g., as `assert` and `print` statements), which is particularly advantageous where properties of concurrent programs are concerned. Another example is in the generation of test cases themselves: some methods of static analysis can construct a *counterexample* when a specification violation is detected, and this capability can be inverted to provide automated test case generation. The task of the test engineer then becomes that of specifying interesting properties to test, rather than constructing the test cases themselves.

To move beyond static analysis of local properties and toward full program verification—that is, toward analysis of properties that get to the essential purpose of the program, or to critical attributes of its operation, such as security or safety—requires specification methods capable of describing system requirements, designs, and implementation issues, together with methods and tools for verifying correctness across several levels of hierarchical development. Suitable specification languages and associated methods and tools for formal verification do exist (mostly associated with interactive theorem provers such as our own PVS) but they require an investment of skill and time that currently limits their use to research environments and certain regulated classes of applications, such as the highest "Evaluation Assurance Levels" (EALs) for secure systems. Instead, approaches that seem most promising for wide deployment are those attached to model-based design environments such as Esterel/SCADE, Matlab/Simulink/Stateflow, or UML. These environments provide graphical specification notations based on concepts familiar or acceptable to engineers (e.g., control diagrams, state machines, sequence charts), methods for simulating or otherwise exercising specifications, and some means to generate or construct executable programs from the models. Until the advent of model-based methods, artifacts produced in the early stages of system development were generally descriptions in natural language, possibly augmented by tables and sketches. While they could be voluminous and precise, these documents were not amenable to any kind of automated analysis. Model-based methods have changed that: for the first time, early-lifecycle artifacts such as requirements, specifications, and outline designs have become available in forms that are useful for mechanized formal analysis.

Formal analysis can be used for model-based designs in similar ways to programs—for example, through extended typechecking and other methods of static analysis—but, because model-based descriptions can be higher-level and more abstract than programs, it is often possible to explore properties that are closer to the real intent or requirements for the design. Furthermore, because model-based designs are explored through simulation, they generally include a model of the environment (e.g., the controlled plant, in the case of embedded systems) and thereby provide a more complete foundation for formal analysis than an isolated program. In many cases, the environment model is a *hybrid system* (i.e., a state machine with real-valued

variables and differential equations) and the properties of interest involve real time. Formal analysis of hybrid and timed systems is challenging, but is becoming effective.

Intermediate-level requirements are often stated informally as "shalls" and "shall nots" (e.g., "the microwave shall not be on when the door is open"), sometimes with temporal constraints added (e.g., "the windshield wipers shall turn off if no rain is detected for 10 seconds"). These can be formalized as invariants—and given a sufficiently rich extended type system (e.g., one that has predicate subtypes) they can even be stated as types—but these approaches require extensions to the modeling notation. Engineers may prefer to stay within their familiar framework, so an alternative approach is to specify requirements as *synchronous observers*: that is, as separate models that observe the state of the system and raise an error flag when a requirement is violated (rather like an `assert` statement in a program). Formal analysis then searches for circumstances that can raise the error flag, or verifies their absence. Counterexamples generated by formal analysis can be used to drive the simulator of the modeling environment, or they can be presented to the user in one of its modeling notations (e.g., as sequence charts). By modifying the observer to recognize interesting, rather than undesired, circumstances, we can use formal analysis to help explore models (e.g., "show me an execution in which both these states are active and this value is zero"), and a variant of this approach can be used to generate test cases at the integration and system levels.

As these examples illustrate, the applications of formal methods in software engineering are becoming increasingly diverse and powerful. We now provide a brief survey of the technology that has enabled these advances.

III. The Technology of Formal Methods

The applications of formal methods outlined in the previous section are made possible by remarkable recent progress in the automation of formal calculations. This progress is not the result of any single breakthrough: it comes from steady progress in a number of basic technologies and from new methods for using these technologies in synergistic combination.

We gave an example earlier where it was necessary to decide if $z + 2$ could be zero when $z = 3 \times x + 6 \times y - 1$, and x, y, and z are integers. A program that can solve this kind of problem is called a *decision procedure*; in this particular case it is a decision procedure for the theory of integer linear arithmetic. Efficient decision procedures are known for many theories that are useful in computer science and software engineering, including the unquantified forms of uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, and arrays; furthermore, these decision procedures can work together to solve problems in their combined theory.

A decision procedure reports whether a set of constraints is satisfiable or not; the example above is unsatisfiable on the integers but satisfiable on the reals—a satisfying instance is $x = -\frac{1}{3}$, $y = 0$, and $z = -2$. A decision procedure that can provide instances for satisfiable problems is called a *satisfiability solver*. Propositional calculus (the theory of "Boolean" variables and their connectives) is a particularly useful theory: it can be used to describe hardware circuits and many finite-state problems (these are encoded in propositional calculus through what amounts to circuit synthesis). A satisfiability solver for propositional calculus is known as a SAT solver, and modern SAT solvers can deal with problems having millions of variables and constraints. A satisfiability solver for the combination of propositional calculus with other theories is called an SMT solver (for Satisfiability Modulo Theories), and a recent surge of progress (spurred, like that of SAT solvers, by an annual competition) has led to SMT solvers that can handle large problems in a rich combination of theories.

Many formal calculation tasks can be formulated in terms of SAT or SMT solving. For example, to discover if a certain program or executable model can violate a given assertion in k or fewer steps (where k is a specific number, like 20), we translate the program or model into a state machine represented as a transition relation over the decided theories, conjoin the k-fold iteration of this relation with the negation of the assertion after each step, and submit the resulting formula to an SMT solver. A satisfying instance provides a counterexample that drives the program to a violation of the assertion. This approach is called *bounded model checking* and a modification (essentially, where the assertion is not negated) can be used for the generation of test cases or exploration instances. Invariants can be verified using a different modification called *k-induction*. Unfortunately, k-induction is not a complete procedure and its complexity is exponential in k, so there are many true invariants that cannot be verified in this way. One way to strengthen it is to supply additional invariants. These could be suggested by the user but often a large number of rather trivial

auxiliary invariants are needed and it is unreasonable to expect the user to suggest them all: instead, they must be discovered automatically.

One class of methods to generate or prove invariants uses abstraction: that is, aspects of the program or model are approximated or simplified in some way in the hope that analysis will thereby become more tractable without losing so much detail that the desired property is no longer true. For example, we could replace an integer variable by one that records only its qualitative sign (i.e., `negative`, `zero`, or `positive`) and likewise abstract the operations on the variable so that, for example, `negative` + `negative` yields `negative` but `negative` + `positive` nondeterministically yields any of the three sign values. Then we simulate execution of the program to see if any of the abstracted variables converges to a fixed point, thereby yielding an invariant. This method of *abstract interpretation* is sensitive to the abstraction chosen, to the way in which abstracted values are combined when different execution paths join, and to the "widening" method used to accelerate or force convergence to fixed points. Abstract interpretation is often used on its own as a method of static analysis; using it to generate "helper" invariants for other methods is an example of an increasingly powerful trend: the use of many methods in combination.

In addition to generating invariants, abstractions are also used in verifying them: because the abstracted system should be simpler, or have fewer states, than the original, a verification method that is defeated by the scale of the original may succeed on the abstraction. *Predicate abstraction* is often used for this purpose; whereas abstract interpretation approximates the values of variables, predicate abstraction approximates relationships between variables. For example, we may eliminate two integer variables x and y and replace them by a trio of Boolean variables that keep track of whether or not $x < y$, $x = y$, or $x > y$. Decision procedures or SMT solvers are used in construction of the abstracted system corresponding to a given set of predicates. The resulting system will have a finite number of states so it becomes feasible to compute those that are reachable—perhaps by *explicit state model checking*, which is essentially exhaustive simulation, or by *symbolic model checking*, which usually employs a potent data structure called reduced ordered binary decision diagrams, or BDDs. Invariants are easily checked once the reachable states are known, and more general properties can be checked through translation to Büchi automata. Methods for abstraction and reachability analysis in timed and hybrid systems can be developed along similar lines, as we illustrate in the next section.

IV. A Simple Example

We illustrate several of the themes and technologies described above with an example originally due to de Oliveira and Cugnasca.[1] The example concerns conflict detection in aircraft routing and is massively simplified, since its purpose is simply to illustrate the methods.

Aircraft routes are specified in this example as a sequence of "windows"; these are two dimensional regions in space through which the aircraft is required to pass in the order specified. Between two windows, the aircraft holds its speed and heading within a specified range. When an aircraft passes through the next window in its path, it is given new ranges for its speed and heading. A particular window can appear in the route of several aircraft and it is possible for this confluence of routes to lead to a "conflict" where the separation between two or more aircraft falls below a specified minimum. The task is to model this system and to predict any potential conflicts arising in a given scenario.

This example has both discrete and continuous components: when an aircraft passes through a window, its behavior undergoes a discrete change and it enters a new mode or state; within each mode, an aircraft's behavior is modeled by its velocities in each of the modeled dimensions, which are continuous quantities. This is therefore an example of a *hybrid system*.

The specific scenario we are given is illustrated in Figure 1. This scenario simplifies the problem by reducing the space dimensions from three to two, so that the windows become line segments. The example involves two aircraft a and b whose initial positions are $(-4, -3)$ and $(-1, 0)$, respectively. There are two windows; Window 1 is specified by $x = 0$, $-1 < y < 2$ and Window 2 by $x = 4$, $-1 < y < 1$. The velocity of aircraft a is denoted `vxa` and `vya` for the x and y dimensions, respectively, while that for aircraft b is similarly denoted `vxb` and `vyb`. Both aircraft are heading toward Window 1 and their velocities are constrained by $1 <$ `vxa` < 3 and $-1 <$ `vya` $< -\frac{1}{4}$ and `vxb` $= 2$ and `vyb` $= 0$ (i.e., aircraft b is flying along the x axis at a constant velocity). After passing Window 1, the constraints on aircraft a change to $1 <$ `vxa` < 3 and $-1 <$ `vya` < 0, while those for aircraft b remain the same.

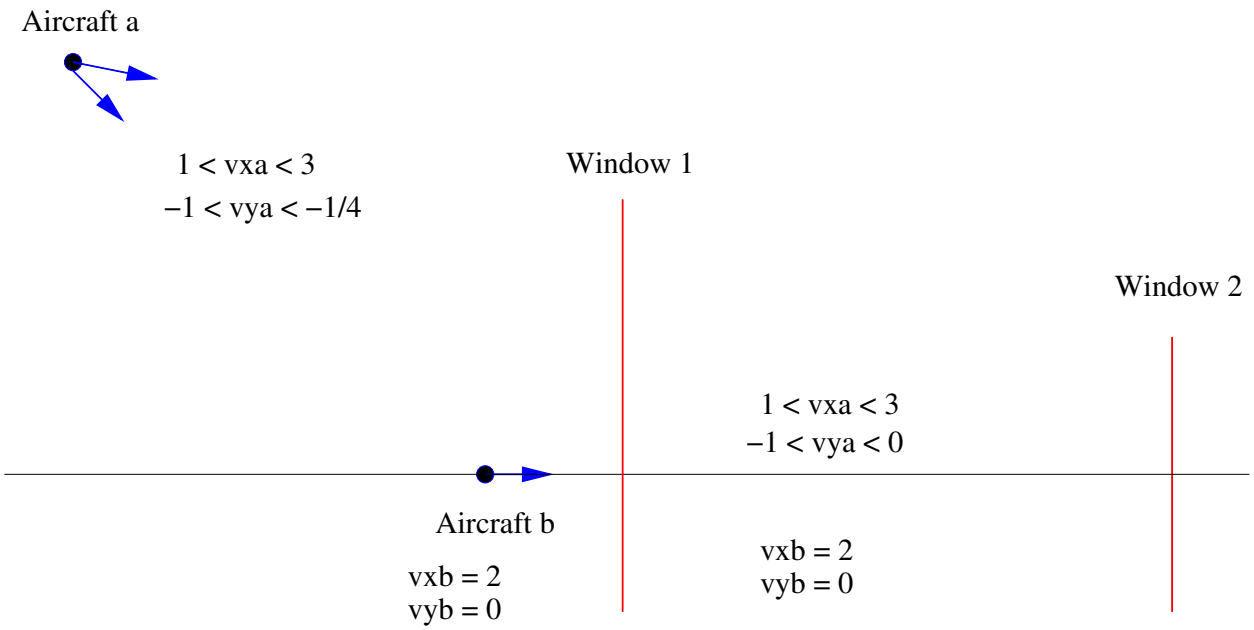

Figure 1. A Simple Two-Aircraft Routing and Conflict Scenario

For formal analysis, we specify this example in the language of Hybrid SAL, which is one of the tools in SRI's SAL suite [a]. In Hybrid SAL, variable names ending in `dot` refer to the derivative of the variable with the same name minus `dot`; thus, we will use `xa` to represent the position of aircraft a in the x dimension, and `xadot` will be its velocity in that dimension; the variables representing the y dimension and those for aircraft b are named similarly. These variables represent the actual positions and velocities of the two aircraft; at each instant, the environment can select new values for the x and y velocities of aircraft a, subject to the constraints imposed by the current mode (i.e., which window it is flying toward): these are represented by the variables `vxa` and `vya`. The current mode is represented by the discrete (actually, Boolean) variable `Window2`, which is `TRUE` when aircraft a is flying toward Window 2, and `FALSE` when it is flying toward Window 1. These elements of the specification are shown in the following box.

```
twoaircraft: MODULE =
BEGIN
 LOCAL xa, ya: REAL
 LOCAL xadot, yadot: REAL
 INPUT vxa, vya: REAL
 LOCAL xb, yb: REAL
 LOCAL xbdot, ybdot: REAL
 LOCAL Window2: BOOLEAN
```

We also specify the initial mode and the initial positions of the two aircraft.

```
 INITFORMULA
   Window2 = FALSE
     AND xa + 4 = 0 AND ya = 3 AND xb + 1 = 0 AND yb = 0
```

Next, we specify the dynamics of the system. These are given as a transition relation (i.e., a nondeterministic state machine) specified by guarded transitions. The guard in each transition appears before the `-->` arrow, and the state change appears after it; the primed variables represent values in the new (changed) state and unprimed variables represent values in the old (pre-change) state. Here, we have one discrete transition (an update to a discrete variable), and two continuous ones (one for each mode). As the continuous

[a] Our tools are available at `http://fm.csl.sri.com`.

variables evolve, the hybrid state machine constantly evaluates its guards. If any of them is true, then its corresponding state change is performed atomically.

```
TRANSITION
[  %% Discrete Mode Change
   xa = 0 -->
        Window2' = TRUE
[] %% Continuous evolution in mode toward Window 1
   NOT Window2
    AND xa < 0 AND vxa > 1 AND vxa < 3 AND vya+1 > 0 AND 4*vya + 1 < 0 -->
        xadot' = vxa;
        yadot' = vya;
        xbdot' = 2;
        ybdot' = 0
[] %% Continuous evolution in mode toward Window 2
   Window2
    AND xa >= 0 AND vxa > 1 AND vxa < 3 AND vya+1 > 0 AND vya < 0 -->
        xadot' = vxa;
        yadot' = vya;
        xbdot' = 2;
        ybdot' = 0
]
END;
```

Finally, we specify the property that we hope will be true of this system: namely, that the two aircraft never come too close together. Hybrid SAL uses a form of linear temporal logic (LTL) to specify properties; here, the `G` modality means "always" so the property is specifying that the two aircraft should never get closer than 1 unit in either the x or y dimensions.

```
noconflict: THEOREM
  twoaircraft |- G( NOT( xb - xa < 1 AND ya - yb < 1 ) );
```

Analysis of hybrid system is quite challenging; few methods can handle more than four or five continuous variables. Hybrid SAL uses a novel method that has been able to handle some tens of variables.[2] It works by using theorem proving over real numbers to calculate a conservative approximation to the original hybrid system. The continuous variables of the original are replaced by qualitative signs (i.e., `negative`, `zero`, or `positive`) of polynomials over these variables. The power of the method depends on the number and selection of polynomials employed. Hybrid SAL has effective heuristics for guessing suitable polynomials based on those appearing in the specification (e.g., `4*vya + 1`) but in this case it is necessary to tell it to use an additional one: `2*xa-3*xb+5`.

.

With this hint, Hybrid SAL produces another SAL specification that conservatively approximates the original hybrid system but that is purely discrete. This new specification is too long to show here (520 lines) but it is generated in a fraction of a second. Because it is discrete, this new specification can be analyzed by other tools in the SAL suite—in particular, its symbolic model checker.

In this case, the symbolic model checker quickly generates a counterexample. It is possible that this is due to inexactness introduced by the approximation, so it is necessary to translate the counterexample back from the discrete to the hybrid model and to check whether the scenario can be reproduced there, in Matlab for example (at present, this must be done by hand). Here, the counterexample is real: one instance is where aircraft a flies directly toward $(0, 0)$. Because of the simple trajectory of aircraft b, it is feasible to discover this with pencil and paper, but a little additional complexity will take this example beyond what it is feasible to do by hand. Although this example admits conflicts, we conjecture that they occur only when flying to Window 2. We can specify this conjecture as follows.

```
noconflict_1: THEOREM
  twoaircraft |- G(Window2 OR NOT( xb - xa < 1 AND ya - yb < 1 ) );
```

Hybrid SAL and the SAL symbolic model checker are able to prove this conjecture in less than a second.

V. Status and Prospects

The first applications of formal methods to see widespread industrial adoption are highly automated static analyzers that shield the user from direct contact with the underlying formal method. For example, Coverity and several other companies provide static analyzers for C and some other programming languages. These analyzers are mostly used for bug finding (i.e., they are tuned to minimize false alarms and cannot verify the absence of bugs) but still find many errors in widely used and well-tested software: for example, Coverity found an average of 0.434 bugs per 1,000 lines of code in 17.5 million lines of C code from open source projects.[3] The Spark Examiner from Praxis is tuned the other way: it may generate false alarms but does not miss any real errors within its scope—in particular, it can verify the absence of runtime exceptions in Ada programs. It has found errors in avionics code that had already been subjected to the testing and other assurance methods for DO-178B Level A.[4]

The Astrée static analyzer uses abstract interpretation to guarantee absence of floating point errors (underflow, overflow etc) in flight control software for the Airbus A380,[5] while AbsInt uses abstract interpretation to calculate accurate estimates for worst-case execution time and stack usage.[6] The SCADE toolset from Esterel, also used by Airbus and for other embedded applications, has plugins for analysis and verification by bounded model checking and k-induction.[7] Rockwell-Collins has developed a prototype toolchain with similar capabilities for the Simulink/Stateflow model-based development environment from Mathworks and reports substantial improvements in effectiveness and reductions in cost in its process for reviewing requirements.[8] Mathworks' own Simulink Design Verifier performs test generation, bounded model checking, and verification by k-induction.[9] Xia, Di Vito, and Muñoz have successfully generated MC/DC tests for avionics code with complex arithmetic using techniques based on predicate abstraction and software model checking.[10]

In current work, we are developing and applying formal methods to diagnostic and monitoring systems of the kind used in integrated vehicle health management (IVHM) and to aircraft routing and separation methods. Most of these applications concern hybrid systems and our goal is to greatly increase the automation that can be applied over the best current methods.[11] To this end, we are increasing the power of our tools with new methods for statespace exploration,[12] more powerful methods for SMT solving,[13] further improvements to analysis of hybrid systems,[14] and a "toolbus" that enables several tools to be used in combination.[15]

Acknowledgments

Discussions with Ben Di Vito of NASA Langley Research Center, and funding through NASA Cooperative Agreement NNX08AC59A are gratefully acknowledged.

References

[1]de Oliveira, I. R. and Cugnasca, P. S., "Checking Safe Trajectories of Aircraft Using Hybrid Automata," SafeComp *2002: Proceedings of the 21st International Conference on Computer Safety, Reliability, and Security*, edited by S. Anderson, S. Bologna, and M. Felici, Vol. 2434 of *Lecture Notes in Computer Science*, Springer-Verlag, Sept. 2002, pp. 224–235.

[2]Tiwari, A., "Abstractions for Hybrid Systems," *Formal Methods in Systems Design*, Vol. 32, 2008, pp. 57–83.

[3]*Scan home page*, `http://scan.coverity.com/`.

[4]German, A., "Software Static Code Analysis Lessons Learned," *Crosstalk*, Nov. 2003, Available at `http://www.stsc.hill.af.mil/crosstalk/2003/11/0311German.html`.

[5]*Astrée home page*, `http://www.astree.ens.fr/`.

[6]*AbsInt home page*, `http://www.absint.com/`.

[7]*SCADE Design Verifier*, `http://www.esterel-technologies.com/products/scade-suite/design-verifier`.

[8]Miller, S. P., Tribble, A. C., and Heimdahl, M. P. E., "Proving the Shalls," *International Symposium of Formal Methods Europe, FME 2003*, edited by K. Araki, S. Gnesi, and D. Mandrioli, Vol. 2805 of *Lecture Notes in Computer Science*, Springer-Verlag, Pisa, Italy, March 2001, pp. 75–93.

[9]*Simulink Design Verifier home page*, `http://www.mathworks.com/sldesignverifier/`.

[10]Xia, S., Di Vito, B., and Muñoz, C., "Toward Automated Test Generation for Engineering Applications via Predicate Abstraction," *20th ACM/IEEE International Conference on Automated Software Engineering (ASE'05)*, IEEE Computer Society, Long Beach, CA, Nov. 2005, pp. 283–286.

[11]Muñoz, C., Carreño, V., and Dowek, G., "Formal Analysis of the Operational Concept for the Small Aircraft Transportation System," *Rigorous Engineering of Fault-Tolerant Systems*, Vol. 4157 of *LNCS*, 2006, pp. 306–325.

[12]Ciardo, G., Jones, R. L., Miner, A. S., and Siminiceanu, R., "Logic and Stochastic Modeling with SMART," *Performance Evaluation*, Vol. 63, 2006, pp. 578–608.

[13]Dutertre, B. and de Moura, L., "A Fast Linear-Arithmetic Solver for DPLL(T)," *Computer-Aided Verification, CAV '2006*, Vol. 4144 of *Lecture Notes in Computer Science*, Seattle, WA, Aug. 2006, pp. 81–94.

[14]Gulwani, S. and Tiwari, A., "Constraint-Based Approach for Analysis of Hybrid Systems," *Computer-Aided Verification, CAV '2008*, Vol. 5123 of *Lecture Notes in Computer Science*, Springer-Verlag, Princeton, NJ, July 2008, pp. 190–203.

[15]Rushby, J., "Harnessing Disruptive Innovation in Formal Verification," *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, edited by D. V. Hung and P. Pandya, IEEE Computer Society, Pune, India, Sept. 2006, pp. 21–28.